

A static analyzer for finding dynamic programming errors

William R. Bush_, Jonathan D. Pincus and David J. Sielaff

summary by Minkyung Cho

Traditional Checking: error-checking about static expression of a program

- a. syntax error
- b. type violation
- c. mismatch between a function formal and actual parameter
- d. pointer analysis (but for code generation)

1. Most Analysis -> code optimization (compiler)

Three significant restriction

1. **inter-procedural** -> OK
 - But interactions between function are not detected.
2. not applicable to many categories
 - ex) **memory leaks**
3. for code generation
 - **not for error-checking**

2. Lint(specialized Checkers) : code check

Broader classes of errors -> such as stylistic mistake and potential portability problems

- Still limited and are prone to reporting **false errors**.
- Report potential problems on **non-achievable paths**

3. Annotation Checker (LCLint, Aspect, The Extended Static Checker and monadic second-order logic checker)

Require substantial additional work by the user

4. Abstract interpretation

Analysis must be correct for all possible inputs

5. Purify (*: debugging tool)

Detect a broad range of errors and require no extra programmer effort to use.

However, debuggers, operating on **heavily** instrumented executables **requiring test cases**, which impose serious limitations.

GOAL

- to develop a source code analyzer that could find Purify-like errors with Purify's ease of use, but without needing test cases.

Requirement

1. Real world programs written in C and C++ should be checked effectively.
2. Information should be derived from the program text rather than acquired through user annotations.
3. Analysis should be limited to achievable paths; that is, sequences of program execution which can actually occur in practice.
4. The information produced from the analysis should be enough to allow a user to characterize the underlying defects easily.

PREfix's KEY CONCEPT

1. Simulation: Use Virtual Machine => Memory Test & restrict *achievable* path.
2. The behavior of a function : Model
3. Model is automatically generated by information extracted from programs.
4. Bottom-Up: can apply an entire program, or subset of a program.

Warning Message Description

1. filename, function name, #line
2. the type of defect (ex. Memory leak)
3. variable name
4. declared where
5. which models of called function was used (system library or user define)
6. error trace path
7. the conditions on that path that caused the defect

Process Diagram

Parsing (use standard C/C++ compiler front end)

➔ AST

➔ Ordering (topological sort : functions)

➔ Loading exist models (library or before user models)

Simulation Using Virtual Machine

```
while (there are more paths to simulate){
  initialize memory state
  simulate the path, identifying inconsistencies and updating the memory state
  perform end-of-path analysis using the final memory state,
  identifying inconsistencies and creating per-path summary
}
combine per-path summaries into a model for the function
```

Figure 5. Pseudo-code for function simulation.

1. Simulating achievable path

- Since the number of achievable paths through a function is still often quite large, PREFIX selects a representative sample of achievable paths to simulate. The maximum number of paths in that sample is a [user configurable setting](#).

2. Memory: exact values and predicates

- Memory Layout,
- Bit Operation
- Dereference

3. Conditions, assumptions and choice points

```
4 char *f(int size)
5 {
6 char *result;
7
8 if (size > 0)
9 result = (char *)malloc(size);
10 if (size == 1)
11 return NULL;
12 result[0] = 0;
13 return result;
14 }
```

We don't know the value of "size".

Assumption: size > 0

Condition: 8: if (size > 0) => false

Choice points: 10: if (size == 1) (need to make assumption : size == 1, size != 1)

Assumption: size <= 0

Condition: 8: if (size > 0) => false

10: if (size == 1) => false

Choice points:

The simulator picks one choice for the current path, and investigates the alternate choices on subsequent paths.

4. End-of-path analysis

During end-of-path analysis, the simulator performs a simple mark-sweep operation to discover which memory is reachable from an external. Any unreachable memory or resource that has been allocated, but not freed, is reported. (Notes, Modeling allocation and deallocation of memory requires special model operators.)

Models

MODEL: The behavior of a function: model consists of exclusive outcomes.

Model structure

<OUTCOME>
: *Guards* -> an element of test set (input dependent)
: *Constraints* -> precondition
: *Results* -> postcondition

```
1 int deref(int *p)
2 {
3     if (p==NULL)
4         return NULL;
5     return *p;
6 }
```

Figure 7. A dereferencing function.

```
(deref
  (param p)
  (alternate return_0
    (guard peq p NULL)
    (constraint memory_initialized p)
    (result peq return NULL)
  )
  (alternate return_x
    (guard pne p NULL)
    (constraint memory_initialized p)
    (constraint memory_valid_pointer p)
    (constraint memory_initialized *p)
    (result peq return *p)
  )
)
```

Figure 8. The model of the dereferencing function. (2 outcome exists (1. p == NULL; 2. P != NULL))

Model Emulation

- (i) The appropriate model is retrieved. This may require dereferencing a function-valued variable.
- (ii) All guards are evaluated to determine eligible outcomes. Possible values for a guard are FALSE (the outcome is not eligible for selection), TRUE and DON'T KNOW (some value or values referenced in the guard are unknown).
- (iii) An eligible outcome is selected. This selection creates a choice point.
- (iv) All guards in the selected outcome having unknown values are assumed. This is done in the same manner as boolean expressions, discussed above.
- (v) All constraints are tested and any violations are reported as warnings.
- (vi) All results are evaluated.

Model Generation

System lib -> provided by *Prefix*

Their automatic generation is crucial to the usability of PREFIX as a real-world tool. The basic technique of model generation, or *automodeling*, is to remember the relevant state of memory at the end of each path, and then to merge the disparate states into one model at the end of function simulation.

Merging states at the end of function simulation is not necessary semantically but performance. However, only exactly equivalent outcomes are merged directly. For example, if one outcome had a result `*p=5` and another had `*p=8`, the outcomes were merged with a fuzzy result of '`*p` is initialized'. As a result, any outcome merging approach which lost information was abandoned.

Incomplete knowledge and conservative assumptions

Third party components: source code or models are not available.

The general philosophy is to be conservative: **avoid generating incorrect messages**, even at the potential expense of failing to identify real defects.

For unmodeled functions, this conservatism specifically means assuming the following.

- The function can modify any memory it has access to (all memory pointed to by non-constant parameters is set to an unknown initialized value).
- All pointers are used (all pointers are aliased).
- The return value can be anything (so that later use of the return value will be correct in any context).

In this case the pointer is simply marked as **'lost'** and not reported as leaked. An interesting future possibility is to report this lost memory, which may assist the user in tracking down issues related to overall program memory consumption.

DEVELOPMENT AND USE OF THE ANALYZER

The most important mechanisms in that process have been: path selection (including the heuristic described above), model abstractions capturing just enough detail, model simplification through outcome merging (also described above), and parameterizing analysis so that it can be tuned as required by circumstances.

The model file for a source file typically is roughly the same size as its debug object file, even with simple (exact) outcome merging,

PREfix contrasts sharply with annotation-based ESC, LCLint and ASPECT.

Also note that to be completely correct the analyzer must correctly model the operation of the target hardware (with respect to floating point, for example) and the action of any optimizing compiler used in the build process.

RESULTS

The decrease in coverage

First, models, with their multiple outcomes, cause **additional paths** to be traced through calling functions.

Second, **dead code is revealed when models are used**,

CONCLUSIONS AND OBSERVATIONS

The vast majority of defects identified by PREfix are due to the interaction between multiple functions;

- PREfix tends to find errors off main code paths.
- Different coding styles and disciplines result in greatly different error rates

Limitations

As mentioned above, modeling has been driven by pragmatic concerns. The language only supports relational constraints, the construction of sets of bits, the points to operator, and successions of dereferences and offsets (which is how naming works and how structures are modeled).

There are obvious limitations of expressiveness in the language as it now stands.

- ➔ <http://chess.eecs.berkeley.edu/pubs/talks/02/neculaResOverview.pdf> (CCure)
- ➔ <http://www.cs.cornell.edu/projects/cyclone> (Cyclone)

